ARMY RESEARCH LABORATORY

A MATLAB Library for Rapid Prototyping of Wireless Communications Algorithms with the Universal Software Radio Peripheral (USRP) Radio Family

by Gunjan Verma and Paul Yu

ARL-TR-6491 **June 2013**

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-6491 **June 2013**

A MATLAB Library for Rapid Prototyping of Wireless Communications Algorithms with the Universal Software Radio Peripheral (USRP) Radio Family

Gunjan Verma and Paul Yu
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* June 2013	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** 1 October 2011 to 30 September 2012

4. TITLE AND SUBTITLE A MATLAB Library for Rapid Prototyping of Wireless Communications Algorithms with the Universal Software Radio Pheriphal (USRP) Radio Family	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Gunjan Verma and Paul Yu	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-CIN-T 2800 Powder Mill Road Adelphi, MD 20783-1197	**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-6491
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Software-defined radios (SDRs) provide researchers with powerful and flexible wireless communications experimentation platforms. This report describes the development and usage of an U.S. Army Research Laboratory (ARL)-developed powerful software library that allows the user to interface with the Universal Software Radio Peripheral (USRP) family of radio hardware using MATLAB. This library enables fast prototyping of algorithms on SDRs.

15. SUBJECT TERMS
USRP, MATLAB, GNU radio

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Gunjan Verma
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	32	**19b. TELEPHONE NUMBER** *(Include area code)* (301) 394-3102

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

INTENTIONALLY LEFT BLANK.

1. Introduction

A software-defined radio system (SDR), is a fully functional radio communication system whose components, e.g., multiplexers, modulators/demodulators, and filters, are primarily implemented in software rather than hardware. Great flexibility is gained with software-heavy paradigms. However, significant programming expertise, time, and patience are often required to work with SDRs.

In this report we first discuss the strengths and weaknesses of the most popular software radio project, GNU Radio. While this is a specific SDR implementation, the discussion is broadly applicable to available SDRs in general.

Then, we present our approach for lowering the barrier to entry for SDR research, especially for the scientist that does not specialize in software development. This effort has resulted in the MATLAB-Universal Software Radio Peripherial (USRP) Library (MUL) that allows researchers to use the popular mathematical software MATLAB to interface directly with the USRP, thus avoiding GNU Radio and its complexity. This report also includes detailed notes on the usage of the software library.

1.1 Challenges of GNU Radio

GNU Radio is a software development toolkit that provides signal processing blocks to drive the SDR. GNU Radio has many strong points – it is actively maintained with a large user base, new capabilities are constantly being added, and compiled `C` code is fast for many real-time applications such as video communications. It also integrates very well with National Instruments' USRP (*1*), the most popular hardware used with GNU Radio.

While GNU Radio significantly facilitates SDR research, extending it to implement new signal processing algorithms is often non-trivial. In particular, the implementation of new signal processing blocks in `C++` requires significant programming expertise and the huge (and constantly growing) GNU Radio application programming interface (API) makes learning the architecture a daunting task, even for the experienced software developer. This requirement poses many challenges for the typical research scientist. In our experience, implementing a novel signal processing algorithm in GNU Radio can take months.

GNU Radio also relies on external libraries such as `Boost` (*2*) or `GNU Scientific Library` (*3*) for much of its signal processing due to lack of native `C++` support. This also adds to the difficulty in mastering the GNU Radio architecture. Below, we briefly touch on the difficulties in extending GNU Radio to implement new algorithms.

1.1.1 Complex Software Architecture

As an illustration of the complexity of GNU Radio, consider a simple example. Suppose we want to write a new signal processing routine with GNU Radio in which we conjugate multiply vector a by the conjugate of vector b and store the result in a third vector c. (It turns out that this is actually implemented in `/gnuradio-3.6.0/gnuradiocore/src/lib/general/gr_multiply_conjugate_cc.cc` and `/volk/include/volk/volk_32fc_x2_multiply_conjugate_3gi2fc_u`, but suppose we are unaware of it.) This simple operation is over 150 lines of code.

In MATLAB, by contrast, this could be implemented as

```
c=a.*conj(b);
```

While a conjugate-multiply block does exist within GNU Radio, this examples serves to illustrate the costs in code development effort that a user might incur in developing a *new* signal processing block within GNU Radio. In addition, as shown in figure 1, the organization of existing blocks is quite complex, and even discovering their existence in the massive and growing GNU Radio library can be a challenge.

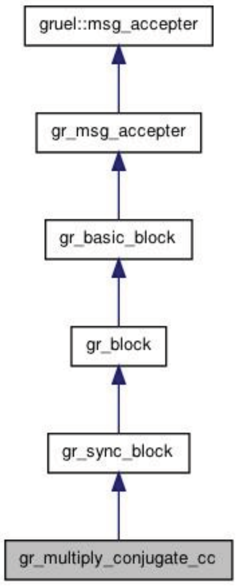

Figure 1. Class hierarchy of the `gr_multiply_conjugate_cc` block (conjugate multiply) in GNU Radio. Arrows denote inheritance from the target block. Note the complexity of the hierarchy even for such a fundamental function, which contributes to the difficulty in finding existing blocks and adding new ones.

1.1.2 Difficult Visualization

C++ and Python are unwieldy when it comes to graphics, which makes visualizations challenging. Figure 2 shows a fast Fourier transform (FFT) plot generated from GNU Radio 3.6. Note that despite the relative simplicity of the figure, the code needed to generate it is over 300 lines of wxPython graphical user interface (GUI) code and uses seven different GUI modules. Each of these lines of code, in turn, calls various complex libraries. In MATLAB, the same plot can be generated simply by

```
plot(fft(Y));
```

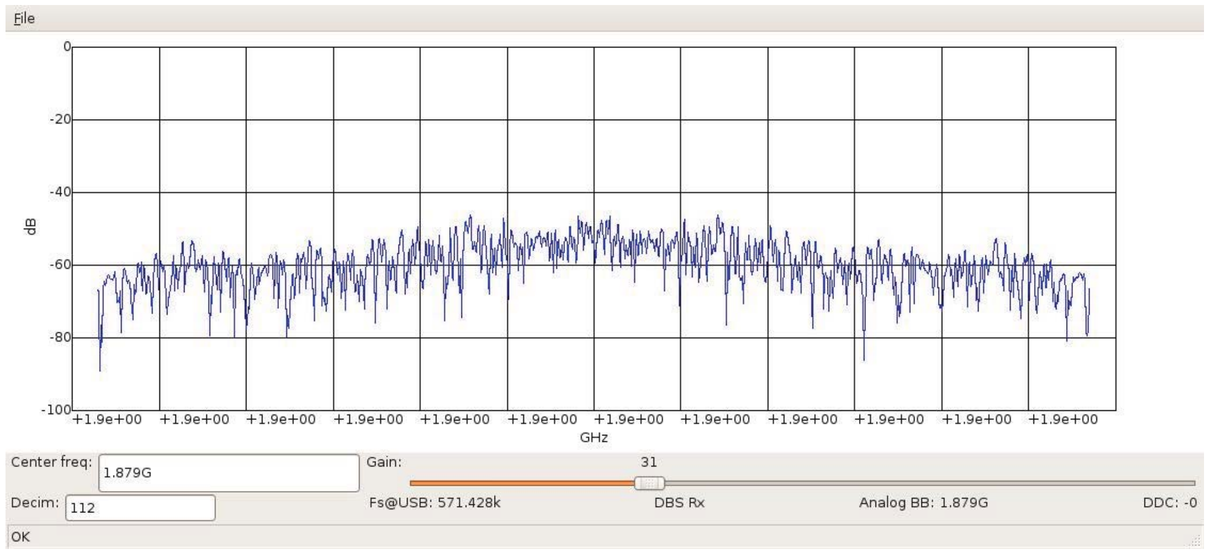

Figure 2. FFT as plotted in GNU Radio. Generating even this simple GUI from within GNU Radio requires hundreds of lines of code to create the background panel, the slider, and the textbox, orient them, and also connect the graph to the underlying data stream.

Another way to visualize data in GNU Radio is to dump it to a file, and import into MATLAB or similar language; while this is easy to do in principle, issues of overhead of inter-process communication, specifically the ability to ensure the visualization can be done in real time, become challenging.

1.1.3 Inadequate Debugging Tools

In GNU Radio, there are two commonly used options for debugging, neither of which are user friendly. The first is to dump data into files in between signal processing blocks, and analyze this in a language like MATLAB. The other is to use the GNU debugger (GDB), stepping through code via breakpoints, and recompiling code. The challenge with the second approach is that proper visualization tools (eye diagrams, constellation plots, etc.) are often needed to enable effective debugging of wireless communications algorithms. By contrast, in MATLAB, debugging is simple because it is intrinsically a scripting language, so values of any variables can be examined on-the-fly and plots can be generated effortlessly.

1.2 Our Approach

We have developed a software library that connects the NI USRP family directly to MATLAB, alleviating the challenges mentioned in section 1.1. However, this approach has a caveat. MATLAB is a high-level language with lower performance than `C++`, and so the achievable real-time data rates are less with MATLAB than with GNU Radio. Nonetheless, in most cases, a working prototype is desirable before investing time to develop a highly efficient implementation.

The MUL provides this ability to rapidly prototype wireless communications algorithms and conduct real-time, over-the-air experiments. In addition, in our experiments, we find that we can achieve data rates on the order of 100 kB/s with MUL.

The main features of the MUL are as follows:

1. Linux-based. This is the platform of choice for much of the scientific community due to its open-source nature. This ensures wide compatibility and easy adoption by the community.
2. Hardware compatibility. MUL works across the entire USRP family, including the USRP 1. Rather than using USB- or Ethernet-specific function calls to move data to/from the radio, the MUL uses the Ettus Universal Hardware Driver which provides a platform-agnostic API to the USRP family.
3. No additional MATLAB toolboxes are necessary. While the provided sample scripts in MUL do use the Communications Toolbox, e.g., for demodulation, there is no inherent dependency to build the library on the target machine. Open-source implementations of most common toolbox functions are readily available.
4. Real-time capability. We have demonstrated real-time transmit/receive capability of MUL to work with Phase Shift Keying (PSK) and Quadrature Amplitude Modulation (QAM) signals at 4 megasamples per second (MSPS). Communicating at this rate includes all necessary transmit/receive chain functions, including automatic (not manual) per-packet frequency offset correction.
5. Packetization capability. We provide a high-performance general-purpose packetization module, which essentially follows the IEEE 802.11 standard for packet

structure. All relevant packet parameters can be specified in MATLAB, while actual packetization-based routines such as rough packet start detection, frequency offset correction, symbol timing acquisition, etc., are implemented in `C` using algorithms based on 802.11, briefly detailed in section 3. Using a 2.7-GHz 64-bit laptop, we have achieved data rates of 100 Kb/s in real time, where we can process 1 second's worth of received data in 0.7 s.

6. Extensibility beyond MATLAB. We have done some initial porting of MUL to Octave (free), and we believe that Octave extensibility is attainable. This would make MUL the lowest-cost solution of its kind, depending only on the USRP and all open-source, free software, allowing it to scale to large SDR testbeds without the cost burden of several per-radio licenses for MATLAB and associated toolboxes.

There are three other major libraries that have sought to connect the USRPs to a high-level language such as MATLAB. Below we briefly review these, along with their relative strengths and weaknesses.

1.2.1 Alcatel-Lucent Library

`Available at http://www.flexible-radio.com/tools4sdr/download`

This library is only available for Windows and only works with the USRP1 and Rx2400 daughterboard. It is no longer maintained actively (last updated in 2009). In addition, the library is delivered as a dynamic link library (DLL), so the code cannot be extended or modified in any way. Finally, a packetization module (i.e., functionality to transmit/receive packets, such as packet synchronization algorithms) is not provided.

1.2.2 National Instruments USRP Library

`Available at http://joule.ni.com/nidu/cds/view/p/id/2679/lang/en`

This library only works with Windows. In addition, it is designed for use with LabVIEW ($2700 per license) and requires several additional toolboxes, such as the Modulation toolkit ($2200), Mathscript module ($500), and Digital Filter Design Toolkit ($1000). In addition, a general purpose packetization module is not provided. An add-on packetization module for text-only data is available at `https://decibel.ni.com/content/docs/DOC-18801`. It supports a maximum performance of approximately 500k samples/sec.

1.2.3 Mathworks SDR Toolbox

`Available at http://www.mathworks.com/discovery/sdr/usrp.html`

The Mathworks SDR toolbox only works with the USRPN210 family (\$2000) and requires Simulink (\$3250) in order to build the library, and all the built-in scripts use the Communications toolbox (\$1350), DSP toolbox (\$1350), and Signal Processing toolbox (\$1000). Also, real-time performance is poor; the provided sample Quadrature Phase-Sift Keying (QPSK) code operates at only 200k samples/second. In addition, the library's performance seems to hinge on pre-correction of frequency offset From the documentation available at the above link: "Due to hardware variations among the USRP^{TM} boards, a frequency offset will likely exist between the USRP^{TM} transmitter hardware and the USRP^{TM} receiver hardware. In that case, perform a manual frequency calibration." Finally, a general-purpose packetization module is not available.

2. The MATLAB-USRP Library

2.1 Sample Code

We provide a simple but illustrative example of how easy it is to use MUL to turn a MATLAB simulation script into a real-time wireless experiment.

Suppose we want to test the effect of different timing synchronizers (such as Gardner vs. Early/Late Gate) on communications performance. The following code is a MATLAB script, taken verbatim from the Mathworks documentation, to accomplish this task.

```
% Initialize some data
    L = 16; M = 8; numSymb = 100; snrdB = 30;
    R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;
    % Design raised cosine filters
    txFiltSpec = fdesign.pulseshaping(L, 'Square root raised cosine', ...
                    'Nsym,Beta', 2*filtDelay, rollOff);
    txFilterDesign = design(txFiltSpec);
    txFilterDesign.Numerator = sqrt(L)*txFilterDesign.Numerator;
    % Create System objects
    hMod = comm.PSKModulator(M);
    hTxFilter = dsp.FIRInterpolator(L, txFilterDesign.Numerator);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
                    'NoiseMethod', 'Signal to noise ratio (SNR)', ...
                    'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = dsp.DigitalFilter('TransferFunction', 'FIR (all zeros)', ...
                    'Numerator', txFilterDesign.Numerator);
    hSync = comm.GardnerTimingSynchronizer('SamplesPerSymbol', L, ...
                    'ErrorUpdateGain', g);
% Generate random data
    data = randi([0 M-1], numSymb, 1);
% Modulate and filter transmitter data.
    modData = step(hMod, data);
    filterData = step(hTxFilter, modData);
% Introduce a random delay.
    delayedData = step(hDelay, filterData, delay);
% Add noise
    chData = step(hChan, delayedData);
% Filter receiver data.
    rxData = step(hRxFilter, chData);
% Time synchronize and estimate the delay from the received signal
[rxData_time_sync, phase] = step(hSync, rxData)

[Source: http://www.mathworks.com/help/comm/ref/comm.gardnertimingsynchronizerclass.html
```

The preceding code is representative of most simulations for wireless channels. First, modulate some data, pulse shape it, add channel effects like noise and delay. Then "receive" the data, and do some benchmarking, such as a bit error rate (BER) computation after demodulation. Because the above code is a simulation, both the receive and transmit functions are done in the same script. Now consider transforming this code into a real experiment with the USRPs, using MUL. The below code snippet shows the receiver-side portion of the code; the transmit code side would be similar. We have prefixed with > the additional lines needed beyond the simulation. Also note the absence of the code that adds channel effects, since now there is an actual channel between the transmitter and receiver. Note the overall similiarity to the simulation code, with most lines identical, and the relative compactness of the code.

```
>   % Define parameters and variables related to signal reception
>   Rx_Dedicated_Define_Parameters
>   % Set up workspace variables; pre-allocated for efficiency
>   Rx_Dedicated_Setup_Workspace
>   % Start the USRP to receive symbols into MATLAB
>   Rx_Dedicated_USRP_Start

    % Initialize some data
    L = 16; M = 8; numSymb = 100; snrdB = 30;
    R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;

>   SAMPLE_RATE=4000000

    % Design raised cosine filters
    txFiltSpec = fdesign.pulseshaping(L, 'Square root raised cosine', ...
                    'Nsym,Beta', 2*filtDelay, rollOff);
    txFilterDesign = design(txFiltSpec);
    txFilterDesign.Numerator = sqrt(L)*txFilterDesign.Numerator;
    % Create System objects
    hMod = comm.PSKModulator(M);
    hTxFilter = dsp.FIRInterpolator(L, txFilterDesign.Numerator);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
                    'NoiseMethod', 'Signal to noise ratio (SNR)', ...
                    'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = dsp.DigitalFilter('TransferFunction', 'FIR (all zeros)', ...
                    'Numerator', txFilterDesign.Numerator);
    hSync = comm.GardnerTimingSynchronizer('SamplesPerSymbol', L, ...
                    'ErrorUpdateGain', g);

>   % Receive signal OTA
>   yrx = fread(receive_pipe_handle, 2*SAMPLE_RATE, 'float32');
>   chData = yrx(1:2:end) + yrx(2:2:end)*1i


% Filter receiver data.
    rxData = step(hRxFilter, chData);
% Time synchronize and estimate the delay from the received signal
    [rxData_time_sync, phase] = step(hSync, rxData)

>   % Stop the USRP to receive symbols into MATLAB
>   Rx_Dedicated_USRP_Stop
```

2.2 Preliminaries

Many of the commands used in the library, behind the scenes, require root access, for example, in order to increase system buffer sizes. In order to permit these, the Linux user should be added to the superuser file. Run the command `sudo visudo` and once in the editor, add the line

```
gnuradio ALL=NOPASSWD:ALL
```

to the file, where `gnuradio` is replaced by the Linux user name that will be using the MATLAB-USRP Library.

In addition, in the MATLAB-USRP library's `Core/` subfolder, there is a `C` file called "`configure_pipe.c`". This `C` file should be compiled for the target architecture using a GNU C Complier (GCC). A precompiled binary for 64-bit Linux systems is provided. Also, in the library's `Scripts/`folder, there are three `.c` files that implement packet detection and equalization functions and expose them to MATLAB as function calls. Each of these `C` files should be compiled from within MATLAB through the command

```
mex file_name.c
```

where file_name is replaced by the name of each `C` file. Already mex-compiled versions of the `.c` files in the `Scripts/` folder are provided for 64-bit Linux architecture and MATLAB R2011a.

Finally, all Python files in the `Core/` and `Sample_Files/` subfolders should have their executable flag enabled. However, sometimes permission flags are not preserved across copying operations. If the Python files in these folders are not executable, simply issue the command

```
chmod 777 *.py *.pyc
```

in the `Scripts/`folder.

MATLAB use its own GNU C Library (glibc), which may create problems when trying to communicate with the USRP via UHD. To solve this problem, one can force MATLAB to use the standard glibc, by issuing the following commands (adjust appropriately depending on the target location of MATLAB and the system location of `libstdc++.so.6`).

```
cd /usr/local/MATLAB/R2011a/sys/os/glnxa64
sudo mkdir old
sudo mv libstdc++.so.6* old
sudo ln -s /usr/lib/x86_64-linux-gnu/libstdc++.so.6 libstdc++.so.6
```

2.3 Other

The analog-to-digital (A/D) sample rate is 64 MS/s for the USRP1 and 100 MS/s for the USRPN210. However, the maximum effective rate is limited by the interface bandwidth between the USRP device and the host machine, about 32 MB/s for USB 2.0. It may also be limited by the processing power of the host machine. For example, even though the Gigabit Ethernet interface of the USRP N210 can support 25 MS/s, the host machine likely cannot perform any meaningful digital signal processing (DSP) operations fast enough to keep up.

Also, note that since each complex sample is really two 16-bit quantities, a single complex sample is 32 bits, or 4 bytes. Thus, taking the USRP1 as an example, the theoretical maximum possible sample rate is 8 MS/s (32 MB/s / 4 bytes). However, in our experience, USB 2.0 cannot continuously sustain this speed; thus, it is prudent to use sample rates that are slightly below the theoretical maximum.

3. Implementation Details

3.1 Directory Structure

MUL is implemented across six subfolders:

1. `Core/` Key library files that connect MATLAB and USRP together.
2. `Data/` Recorded baseband complex signal of known high-quality data that can be used when USRPs are unavailable or to test new code against.
3. `Logs/` Various log files that are updated each time MUL is invoked with status information about the pipe and USRP.
4. `Old_Files/` Collection of various files used in earlier versions of MUL or for testing purposes. These can be studied for additional examples of how to use MUL.

5. `Sample_Files/` Several key GNU Radio scripts that can be used to ensure USRP and/or GNU Radio functionality, for example, contains a signal generator and FFT plotter. Not necessary, but useful to check radio behavior independently of MUL.

6. `Scripts/` Contains all MATLAB scripts that use MUL to accomplish some communications task, such as signal transmission, reception, or packet creation. Also contains `C` code that supports these tasks.

3.2 Support Scripts

A transmitter or receiver implemented with MUL begins with the invocation of three support scripts. Here we briefly describe each of them from the perspective of the receiver. The transmitter support scripts are completely analogous.

3.2.1 Rx_Dedicated_Define_Parameters

This support script contains relevant parameters that govern the communications settings of the receiver and creates key system objects:

1. USRP relevant parameters: sample rate (number of samples per second to acquire from the USRP), center frequency and gain, how many seconds to receive data for, and path to log file to output informational and error messages

2. Inter-process communications based parameters (to allow the USRP and MATLAB to communicate with one another): pipe name and pipe buffer size

3. Pulse shaping and matched filtering parameters: Oversampling factor, roll off of pulse and filter delay

4. Time synchronization parameters: Timing synchronizer object (e.g., Early/Late Gate)

5. Packetization-based parameters: length of packet prefix and number of times it repeats per packet, number of symbols per packet, packet symbol timing search range and sliding window size

6. Demodulation parameters: Number of unique symbols in the constellation and demodulation system object

3.2.2 Rx_Dedicated_Setup_Workspace

This script pre-allocates all the major data structures used in MUL; pre-allocation saves a great deal of time by prevent dynamic re-sizing of data objects. In MUL, we assume that we acquire 1 second's worth of data from the USRP, and then process this data (such as searching for packets, demodulating, etc.) and repeat. The approach taken in MUL is to create all data objects to their maximum theoretical size (with respect to 1 s worth of receive data). For example, if the maximum number of packets we can acquire in one second is 1,000, then relevant vectors storing packets are created of size 1,000. If the actual number of packets found is, say, 500, then we only use elements 1 through 500 of the relevant vectors. In summary, we create the maximum possible size data structures, and then subset them with the actual size. This keeps data structures of a fixed size throughout execution, preventing costly dynamic re-sizing and re-allocation.

1. Signal reception workspace variables: complex vector of received data symbols
2. Packetization workspace variables: packet decision statistic, vector of locations in the raw data with exact packet start locations, vector of estimated frequency offsets (one per packet) and packet prefix
3. Match filtering and pulse shaping workspace variables: vector to store match filtered symbols
4. Equalization workspace: vector of frequency offsets, and vector of phase offsets, time/frequency/phase synchronized symbol vector
5. Demodulation workspace variables: complex vector of demodulated symbols and matrix of packets (one column per packet)

3.2.3 Rx_Dedicated_USRP_Start

This script establishes a connection between the MATLAB and the USRP by creating a named pipe, starting the USRP to acquire symbols into this pipe and returning a handle ("receive_pipe_handle"), which allows MATLAB to read these symbols from the pipe. This script also launches an important function called "configure_pipe", whose important role is to increase the default Linux pipe size from 4 KB to a larger size specified by `PIPE_BUFFER_SIZE` in `Rx_Dedicated_Define_Parameters`. (This change in the pipe size is transient, i.e., does not persist between re-starts of the machine). The ideal pipe size should, at a minimum, be large enough to hold 1 second's worth of complex symbols. For example, if the sample rate if 4,000,000, and since each complex symbol takes 16 bytes, a `PIPE_BUFFER_SIZE` of 64 MB is appropriate.

3.3 Data Reception

In order to minimize the use of for-loops, we follow the common theme in real-time processing of "stream-based" operations; that is, we perform vectorized operations on a large chunk, or stream, of data at a time. Concretely, MUL reads in 1 second's worth of over-the-air data at a time, processes it, then reads in the next 1 second's worth of data. In principle, this can be changed to any other time interval; for example, we could read in 0.25 second's worth of data at a time. The advantage of making this value large is that it minimizes the overhead of loops (for example, reading in 0.25 second's of data means we loop 4 times to process 1 s of data; reading in 1 s of data involves only one loop for the same overall data processing), for which MATLAB is notoriously slow.

Data reception is a trivial call to the fread command, which reads in samples from the named pipe into MATLAB. The USRP delivers the data in interleaved in-phase (I/Q) form. We form the complex baseband signal using the command

```
yrx_complex = yrx(1:2:end) + yrx(2:2:end)*1i;
```

where `yrx` is the vector holding the interleaved I/Q samples.

3.4 Matched Filtering

At the receiver, the first step is to match filter the data in a manner that corresponds with the transmitter's pulse shaping. For example, if the transmitter did root raised cosine (RRC) pulse shaping, then the same should be done at the transmitter. This is done easily in MATLAB using the Communications Toolbox pulse shaping commands.

3.5 Packetization

Most modern communications systems use packets to exchange data. Our implementation follows, in spirit, the wireless local area network (WLAN) 802.11a specification for packet detection and frequency offset correction. There are obviously other ways to perform these and other receiver functions, so our approach is modular and easy to modify to suit a particular packet specification. The transmitter has a simple script called createPackets, which inputs the data to transmit, the number of symbols per packet, and a packet prefix (such as a Barker code) to prepend to each packet, and packetizes the data. In our attempt to make the library as general as possible, there is no restriction on the packet structure other than the presence of a string of symbols (such as a Barker code) that repeats at least twice; the receiver is able to use this for an autocorrelation computation to detect a packet start.

At the receiver, the general scheme is to read in a chunk of (say, 1,000,000) complex baseband symbols from the USRP at a time, store these in a buffer, process them, and repeat. Upon reading in the first chunk of complex baseband symbols, the first step the receiver does is to look for approximate packet starts for all potential packets in the stream. There are many ways to do this, such as looking for a spike in received signal energy. We provide an implementation based on the repetitive nature of the preamble and use the delay-and-correlate algorithm to compute a sliding window statistic. The algorithm cross-correlates the received signal with a delayed version of the signal, normalized by the received signal energy in the cross-correlation window. Denoting the periodicity in the preamble delay by D and the window size by L, the relevant equations are given below.

$$a_n = \sum_{k=0}^{L-1} Y_{n+k} Y^*_{n+k+D} \tag{1}$$

$$b_n = \sum_{k=0}^{L-1} Y_{n+k+D} Y^*_{n+k+D} \tag{2}$$

$$c_n = \frac{|a_n|^2}{b_n^2} \tag{3}$$

The packet decision statistic given in equation 3 grows to values near 1 when a repetitive sequence with periodicity D is detected. Therefore, we can consider large values of this statistic as approximate packet start locations.

3.6 Frequency Offset Estimation

A major issue with the USRP's, especially the USRP1 family, is that of frequency offset; we have found frequency offsets as large as 45 kHz between a pair of radios. This large of a frequency offset renders even built in GNU Radio benchmarks ineffective, even though they have synchronization mechanisms such as phase-locked loops built in. In general, there are a variety of techniques that can be used to do frequency offset correction; the approach we take in our MATLAB/USRP library is a data-aided approach based on the presence of periodically repeating sequence of training symbols.

If there is a frequency offset of f_Δ present at the receiver relative to the transmitter, then the n^{th} received complex baseband symbol $y(n)$ relative to the transmitted n^{th} symbol $x(n)$ with sampling instants T is given by

$$y(nT) = x(nT) e^{j2\pi f_\Delta n} \tag{4}$$

Assuming there is a repetitive preamble present (such as the same one used for rough packet start detection), we can compute

$$\begin{aligned} z &= \sum_{k=0}^{L-1} Y_{n+k} Y^*_{n+k+D} \\ &= \sum_{k=0}^{L-1} x(nT)e^{j2\pi f_\Delta nT}(x(nT+DT)e^{j2\pi f_\Delta (nT+DT)})^* \\ &= e^{-j2\pi f_\Delta DT} \sum_{k=0}^{L-1} |x(n)|^2 \end{aligned} \tag{5}$$

Therefore, we can compute the frequency offset as

$$f_\Delta = \frac{\angle z}{2\pi DT} \tag{6}$$

Note that since $\angle z \in (-\pi, \pi)$, as equation 6 makes clear, the maximum estimable f_Δ is a function of the preamble periodicity D and sampling period T; the smaller these latter two are the larger the estimable frequency offset. Based on our observation for the USRP1's that $f_\Delta \approx 40$ kHz, D and T should be small enough (i.e., rapidly repeating preamble and/or high sampling frequency) in order to ensure we can properly estimate the frequency offset. Also note that the variance of f_Δ decreases with window size L; so, for more stable estimates of f_Δ, repeating sequences of larger lengths should be used.

3.7 Symbol Timing

Once we have a rough packet start estimate and use this information to compute the frequency offset and correct the symbols, we need to find the exact sample at which the packet begins. We again use a simple cross-correlation based symbol timing algorithm. The rough packet estimate gives us a packet start estimate to within a few samples. In order to refine this estimate to an exact sample, we proceed as follows:

1. Denote the prefix symbols as P, with length L, and our rough packet start estimate as symbol number n within the data stream. Using the rough frequency offset estimate f_Δ , frequency correct all the symbols in the data stream within the time index $(n, n+L+d)$, where d is some small value, like 10, to account for exact start uncertainty.

2. Compute, for all m in $[0, d]$

$$s_m = \left| \sum_{k=0}^{L-1} Y_{m+n+k} P_k^* \right|^2 \tag{7}$$

3. Compute $l = \arg\max_m s(m)$

Our estimate of the exact packet start is l.

In our implementation, to match the general concept of 802.11a preamble, the preamble is composed of two repeating parts. The first is a repeating Barker code, which we use to find f_Δ. The second is a repeating Hadamard code, which plays the role of P, to acquire symbol timing. Because P is also repeating, it could also be used to refine the frequency offset estimate.

3.8 Timing Synchronization

After match filtering, each packet exists in an oversampled state, for example, a packet of 1,000 symbols exists as 4,000 symbols, if the oversampling factor of the pulse shaper used at the transmitter is 4. Time synchronization is the process of choosing the optimal symbol from multiple samples representing the same symbol. The Communications Toolbox provides several popular algorithms, such as Early-Late Gate or Gardner Synchronizers. Using one of these methods, the process of achieving time synchronization simply involves invoking the synchronizer on each packet.

3.9 Frequency Offset Correction

Section 3.6 detailed how frequency offset is computed on a per-packet basis. We now use this information to correct the frequency offset of each packet. One reasonable possibility is to frequency-offset correct each packet with that packet's individual estimate of frequency offset. However, since frequency offset is relatively constant over the duration of short time intervals, as it is primarily a function of the transmitter and receiver hardware oscillators, a more stable estimate of the frequency offset can be found by averaging across the frequency offset estimates of multiple consecutive packets. Our approach, which is easily modified via MATLAB, is to take the median frequency offset of all packets found in 1 second's worth of data packets and use this to correct the frequency offset of all packets. For situations in which the frequency offset is drifting more rapidly with time, we can instead pool our estimate over shorter time intervals, for example over 0.25 second's worth of packets.

3.10 Automatic Gain Control

Channel attenuation causes the received symbols to typically have lower power than the power of the transmitted symbols. This has a major impact in modulations in which signal amplitude (and not only phase) plays a role, such as QAM or Amplitude Shift Keying (ASK). Our approach to Automatic gain control (AGC) is simple and in the same spirit as 802.11a. For each packet, we compute the average amplitude of the symbols in the packet prefix. (The packet prefix we use, based on Barker codes, is constant amplitude, so the average received amplitude gives us a good estimator of the attenuation factor). For a given packet, denote the value of this average amplitude by A_{rx}. Because the actual packet prefix is known at the receiver, we also know the true (constant) amplitude in the prefix symbols, call it A_{true}. Therefore, we scale the packet by the quantity $\frac{A_{true}}{A_{rx}}$ to restore the received packet symbols to approximately the same average power as the transmitted packet symbols.

3.11 Phase Synchronization

To compute the phase offset of a given packet, the approach is again to simply use the known prefix symbols and compare them against the received prefix symbols. Specifically, let P denote the known packet prefix and let P_{rx} denote the received packet prefix. Letting $P(n)$ denote symbol n of the known packet prefix and $P_{rx}(n)$ symbol n of the received packet prefix, we know that there is an unknown, but nearly fixed phase relationship (since we've already corrected for frequency offset) between the two, i.e.,

$$P_{rx}(n) \approx P(n)e^{j\phi} \tag{8}$$

We compute the quantity

$$A_n = \sphericalangle P_{rx}(n)P^*(n) \approx \sphericalangle |P(n)|^2 e^{j\phi} = \phi \tag{9}$$

and thus our phase estimate is the mean phase over the prefix symbols,

$$\phi = \frac{1}{L}\sum_{1}^{L} A_n \tag{10}$$

We can use this estimate of ϕ to correct the phase of the symbols by multiplying the packet symbols by $e^{-j\phi}$.

A caveat with this estimator is that in equation 9, if $\phi \approx \pi$, then because of the discontinuous nature of the arc-tangent and phase wrapping, successive values A_n may take value either π or $-\pi$. This "toggling" leads to a poor phase estimate in 10. In order to avoid this problem, we instead compute the phase estimate as

$$\phi = \left(\frac{1}{L} \sum_{1}^{L} |A_n| \right) \left(\frac{\sum_1^L A_n}{|\sum_1^L A_n|} \right) \tag{11}$$

Now, if the phase is such that there is "toggling" between π or $-\pi$, equation 11 addresses this by fixing the value to be π. The second term, which is either 1 or -1, simply chooses the right sign.

3.12 Estimating the Quality of Exact Packet Start

The symbol timing acquisition detailed in section 3.7 computes the exact packet start based on maximizing a cross-correlation statistic. It is interesting to note that, we can detect when the exact packet start location is likely to be incorrect, by examining the variance of the vector whose components are A_n given in equation 9. This is because if our packet does not start exactly where we believe that it does, there will be a relative shift between the received prefix symbols and true prefix symbols, which will result in equation 8 failing to hold with a constant (i.e., constant over the prefix symbols) ϕ. In general, there will then by a varying phase relationship between each received prefix symbol and the "corresponding" true prefix symbol, leading to a high variance for the collection of phase estimates A in equation 9. We can detect this situation and take some sort of corrective action. In our experiments, when this situation occurs (rarely, more likely when radios are moving), a reasonable solution is to shift the computed exact packet start location, call it S, computed from the algorithm in section 3.7 to a neighboring value, such as $S + 1$ or $S - 1$, and re-compute the quantities A_n, until a low variance is found.

3.13 Signal Normalization

In modulations where amplitude carries information, such as QAM, the power of a symbol can exceed 1. In our experiments, we find that symbols with power exceeding 1 are transmitted unpredictably by the USRP, perhaps due to clipping. Therefore, it is necessary to normalize the entire transmitted signal so that the peak power of a symbol is less than 1. This is easily done in MATLAB using the `modnorm` function, which amounts to scaling all packet symbols by a constant. At the receiver, we must undo the effect of the normalization, which again is trivially done by dividing the symbols by the same constant used by the transmitter.

3.14 Demodulation

Demodulation is easily done using the Communications toolbox function. We simply invoke the demodulate function on the packets.

3.15 Implementations

As part of MUL, we provide sample transmit/receive codes that are capable of M-PSK and M-QAM communications. This can easily be extended to arbitrary modulation types via suitable MATLAB function definitions. We have successfully tested these scripts with order M of up to 8 for PSK and up to 16 for QAM. The support script `Rx_Plots.m` provides several example codes, which enable one to generate constellations and videos from received data.

4. Performance

There are several techniques that can greatly improve the effective bit-rate of a MATLAB-based real-time radio communications system. Since most of the intensive processing is done at the receiver, it is often sufficient if these techniques are done on the receiver alone.

1. CPU Throttling. Most modern laptops have a built-in driver (such as cpufreqd,) which automatically throttles the CPU and only allows maximum CPU speed when the CPU load exceeds some threshold. During MUL operation, we have found that the CPU may scale back and forth multiple times per second, resulting in inferior performance. Thus, it is usually better to disable CPU throttling altogether or fix it at the maximum possible CPU speed for the course of MUL operation. Though the exact details are outside the scope of this report, one can usually accomplish this either by disabling CPU throttling in the Basic Input/Output System (BIOS) (called speedstep on Intel CPUs), or using a user-space tool (in Linux, for example, one may use cpufreq-set).

2. Buffer size. In streaming data from the USRP to MATLAB, in general, data will flow via USB (Ethernet) into the Linux kernel, and then pass through a pipe (network) buffer before reaching MATLAB. If this buffer is too small, it can be a major bottleneck to the performance of the entire library; hence, always ensure that relevant buffers are large enough. In Linux kernels 2.6.35 and later, named pipes can have their buffers adjusted in two steps; first, edit the `/proc/sys/fs/pipe-max-file` file to specify the maximum buffer size, and second, adjust the actual pipe size (to a value less than the maximum) using the `fcntl` function. (also see `/etc/sysctl.conf`

to adjust network settings; in particular the `net.core.rmem_max = 1000000` and `net.core.wmem_max = 1000000`). Note: The actual syntax used to change the buffer size of a pipe with descriptor `fid`, with a target buffer size specified by `target_buffer_size`, is given by the command: `fcntl(fid, F_SETPIPE_SZ, target_buffer_size)`. Here, `F_SETPIPE_SZ` is an integer constant, which depends on the particular operating system and platform being used; its value is specified in `/usr/include/asm-generic`. Our library's `configure_pipe.c` code contains the commands necessary to adjust the pipe size; it is important to note that modifying pipe size is transient (only lasts as long as the process invoking the `fcntl` command is active and the file pointed by `fid` is open).

3. Thread priority. Background processes (such as virus checkers, mail clients) can occupy resources, which can steal valuable cycles from the receiver and lead to larger variability in receiver performance. In Linux, one can issue the command `nice -n N cmd`, where N is a negative number, to increase the thread priority of the command `cmd`, which, loosely speaking, will schedule the `cmd` process more favorably as compared to other (background) processes with a larger nice value.

4. `C` programming. MATLAB functions that run slowly can be implemented in `C` instead and invoked from MATLAB, using the `mex` compiler. We have done this, for example, for the packet decision statistic computation in the receiver, which does an expensive sliding window, auto-correlation computation. In general, when `for` loops cannot be avoided, `C` coding is much preferable if performance is critical.

5. Parallelization. Several parts of the receiver, such as channel estimation, matched filtering, time/phase/frequency synchronization, and demodulation, can be done on a per-packet basis. With multi-core CPU's being common nowadays, one can potentially exploit these cores by parallelizing parts of receiver operation to yield a significant speedup. For example, one might use MATLAB's Parallel Computing Toolbox for this purpose.

6. GPU Programming. Signal processing operations, such as the FFT, lend themselves well to implementations on graphics processing unites (GPUs); MATLAB's GPU programming capability, also part of the Parallel Computing Toolbox, supports the execution of a large number of functions (such as `fft`, `sin`, `cos`, `filter`, and many others) directly on the GPU. Here, the tradeoff of transferring data from the MATLAB workspace to the GPU and back must be taken into account; nonetheless, significant speeds up are still possible.

5. Conclusions

We have detailed MUL, the MATLAB-USRP Library, an easy-to-use software tool that enables SDR research directly from MATLAB. MUL significantly bridges the gap from a MATLAB simulation script to a real over-the-air experiment, allowing rapid prototyping of wireless communications algorithms, and can sustain data rates on the order of 100 kB/s in real time, on modern laptops.

References

[1] National instruments. http://www.ni.com/usrp/ (accessed October 2012)

[2] Boost C++ Libraries. http://www.boost.org/ (accessed October 2012)

[3] GNU Operating System. http://www.gnu.org/software/gsl/ (accessed October 2012)

List of Symbols, Abbreviations, and Acronyms

A/D	analog to digital
AGC	Automatic gain control
ARL	U.S. Army Research Laboratory
ASK	Amplitude Shift Keying
BER	bit error rate
BIOS	Basic Input/Output System
DLL	dynamic link library
DSP	digital signal processing
FFT	fast Fourier transform
GCC	GNC C Complier
GDB	GNU debugger
glibc	GNC C library
GPUs	graphics processing unites
GUI	graphical user interface
I/Q	in-phase
MSPS	megasamples per second
MUL	MATLAB-USRP library
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
RRC	root raised cosine
SDRs	Software-defined radios
USRP	Universal Software Radio Peripheral
WLAN	wireless local area network

NO. OF COPIES	ORGANIZATION
1 PDF	DEFENSE TECH INFO CTR ATTN DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
6 (PDFS)	US ARMY RSRCH LAB ATTN RDRL CII T R HOBBS ATTN RDRL CIN A KOTT ATTN RDRL CIN T G VERMA ATTN RDRL CIN T P YU ATTN IMAL HRA MAIL & RECORDS MGMT ATTN RDRL CIO LL TECHL LIB